SoC Registers Management: WTO Approach

Bertrand B. Blanc bertrand.blanc@xx.com Xxxxx Xxxxxxxxx – IC Cellular Systems –XXXX SoC Verification Team

Abstract

Handling registers, to program the hardware from the software, involves a huge effort in terms of specification, design, documentation, verification, validation and APIs. The conventional methodologies often result in the creation of multiple sets of register data, each specific to one end-user application. An additional cost, in terms of human and material resources, and timeframes, arises from the use of ad-hoc tactical methods to ensure coherency between the multiple copies of register capture data. We are genuinely facing a lack of unified approach to handle registers globally throughout the design process, from early specifications to application programming.

XXXX-Xxxxxx®, provided by Xxxxx Xxxxxx, allows register information to be captured from register specification that are maintained by owners of functional specs. Once captured, the register data are stored once and for all in a unique database, thereby ensuring consistency, and are used to generate targets for different purposes. The integrity of the register data is checked at the earlier stages before generation, therefore avoiding manual checks, dramatically improving quality, and saving valuable resources. This innovation is a powerful enabler of accurate and coherent register management.

Contents

Introduction										
1	Regi 1.1 1.2 1.3 1.4	Ster Description Format Hierarchy levels Extension Abstraction Constrained Abstraction	2 2 3 4 4							
2	Mod	lel of Computation	5							
3	Prop 3.1 3.2 3.3	Derties Contiguity coherency Range coherency Overlap coherency	5 5 5 6							
4	Exa	mple: IRQ management module	6							
	4.1	Shared library: Generic read / write 1 to clear form declaration	6							
	4.2	Instantiation: Declaration of an IRQ management module	7							
	4.3	MoC: Full instantiated IRQ management component for the FIFO	7							
	4.4	Reuse: Component managing the IRQs of a module .	8							
5	Purj	posed targeted outputs	9							
	5.1	C check library	9							
	5.2	C library	9							
	5.3	Hierarchal C tree	9							
	5.4	E code	10							
	5.5	Documentation output	10							
	5.6	Hierarchal Flat RD-XML	10							
	5./	Gateway to XXXXXX rtl.coni	11							
	5.8	AXXXXXXXX IOF debug	11							
6	Rest	ılts	11							
Su	mma	ry	12							
References										

Introduction

Handling registers correctly is a crucial need within an industrial project. Almost all the involved teams and team members are impacted by registers:

- the integration specification owner needs registers to make IPs communicate with one another
- **the IP specification owner** is expected to define the behaviour of his IP and the way to configure them through a register bank
- **the validation team** uses registers as entry points to test a set of modules within the complete project platform
- the Technical Reference Manual (TRM) team manages this amount of registers to get commented within the manuals
- **the end-user / FAE** uses registers to write the applications according to the features offered by the platform.

Then, managing registers is a valuable effort in order to focus on main industrial criteria: quality policy, processes robustness, reuse and costs reductions. Xxxxx Xxxxx Ltd. [XxxxWs], leader in IP-reuse and register capture, authored the first solution providing an environment able to capture registers at a single place and thereafter compute this database according to the targeted purpose. Registers are entered once by the specification's owners and then generated for both validation and TRM teams.

Xxxx xx toolset was used within a had-hoc flow built over this suite throughout the XXXXXX program. Registers were entered for each IP block, integrated within devices, finally generated for the various purposes in different target languages:

- E code and C library to check the registers in the top-level validation flow: especially access type, power-on-reset values;
- C library to manage validation scripts to access the registers;
- HTML to share the up-to-date register database between the teams;
- RTF dedicated to the TRM team.

Register capture involves four major computer sciences steps built over a semantics, known in the literature as

- **the front-end** to capture the data according to keywords or glyphs fitting the semantics;
- **the checks** to ensure that the captured data are coherent and follow the given semantics;
- **the core algorithms** to compute the captured data from a Tagged Abstract Syntax Tree into a more complex form following a Model of Computation;
- **the back-end** to output some code according to a targeted purpose.

This document proposes two major interleaved ideas:

- (i) a format has been prototyped to address accurately each of the concerns from the front-end to the back-end, through a mathematical semantics and a given Model of Computation.
- (ii) the strengths and the weaknesses of Xxxxx xx have been pointed out and some major improvement identified for future Xxxxx toolset deliveries.

The first part of the document introduces a high level Register Description Format, also called **RD**, to highlight the main front-end features needed to describe registers. The second part briefly presents a Model of Computation, or MoC, to translate the captured registers into a flat model using core algorithms. The third part shows some properties to assess in order to ensure the coherency of the captured registers. The following part is devoted to depict through an example the main topic: Write-Things-Once (WTO). Hence is shown some targeted languages to focus on the back-end. We finally conclude with experimental results.

1 Register Description Format

This draft highlights the main features of a mathematical higher level of register capture and proposes a <u>Model of Computation</u>. This document aims to give an overview of those features through a pseudo-language created for this purpose.

1.1 Hierarchy levels

A register database can be seen as a tree with a root and a hierarchy of nodes deriving each one another. In Xxxxx, this root is called a product, recursively composed of five hierarchal levels:

- 1. a product is composed of instances of cell
- 2. a cell is composed of registers' sets

2004 © Symposium on IC Design Verifi cation

September 2004 - Dallas, TX

- 3. a registers' set is composed of registers
- 4. a register is composed of bit-fields
- 5. a bit-field is composed of enumeration values

These levels define at their respective stage some attributes which tune the registers' database description. These attributes will not be treated in details here, since they are not the main aim of this section. Fixing the levels of hierarchy suffers from flexibility since, for example, a register can be composed of a transient bit-field description, composed of relevant bit-fields.

We have no fixed number levels of hierarchy, but <u>as</u> <u>many levels of hierarchy as needed</u>. We speak about forms composed of four basic attributes and recursive forms to tune this primal form. The six aforementioned hierarchal levels of Xxxxx are defined as a sub-set of this language. A proposal speaks about Xxxxx xx semantics more in details [*BBxxxxS04*].

$$\phi = \alpha \oslash A(\psi_i)_{i \in I_u}$$

A form ϕ is composed of

- an access type α: basically read-only ro, writeonly wo, read-write rw. The special access type all matches all the others when a reduction will be performed;
- ii. an empty set \emptyset . This set will be presented in another section devoted to inheritance;
- iii. a set of final optional attributes A composed of
 - a. a description field δ ,
 - b. an offset field o,
 - c. a width ω setting a contiguous range of significant bits,
 - d. a reset value ρ taken as the current value of the form;
- iv. a set of inner forms $(\psi_i)_{i \in I_u}$ refining the form ϕ . If this set is empty, then ϕ is a final form since it cannot be deeper refined.

A revision register 32-bit wide will be captured by the following piece of code:

```
ro REVISION is
   description "This register contains
        the IP revision code";
   offset 0x0;
   width 32 bit;
   ro reserved is
        description "Read return 0's";
        offset 8 bit;
        reset 0x0;
```

September 2004 - Dallas, TX

```
width 24 bit;
   end reserved;
   ro Rev is
      description "IP revision";
      offset 0x0;
      ro Minor is
         description "Minor Revision";
         offset 0 bit;
         reset 0x1;
         width 4 bit;
      end Minor;
      ro Major is
         description "Major Revision";
         offset 4 bit;
         reset 0x0;
         width 4 bit;
      end Major;
   end Rev;
end REVISION;
```

This piece of code will not be commented because the syntactical keywords have been chosen to have an intuitive meaning. We can notice that the inner form *Rev* refining the definition of the register *REVISION* is an upper-level definition of the revision Id composed in fine of a *Major* and a *Minor* bit-fields each 4-bit wide. In Xxxxx xx, this level is not captured: the revision Id can only be composed of a couple of bit-fields *Minor* and *Major*.

We also introduce some sets in order to be able to define further some relations or applications between stable sets:

- i. Γ_{ϕ} is composed of forms captured by the designer
- ii. $\Gamma_{\langle\phi\rangle}$ is a set of template forms captured by the designer. These forms will be presented in the next section devoted to instantiation of abstract forms
- iii. $\Gamma = \Gamma_{\phi} \cup \Gamma_{<\phi>}$ is the union of the couple of precedent sets
- iv. Φ is the set of all valid forms: $\Gamma \subset \Phi$
- v. Γ^{ϕ} is the set of inner forms of ϕ : $\Gamma^{\phi} = (\psi_i)_{i \in I_u}$

1.2 Extension

We have seen above that a form ϕ declares after the access type α a set which was empty. This set $(\epsilon_i)_{i \in I_m}$ defines a set of forms in Γ to get inherited in order to tune the form ϕ . This artifact contributes to write-things-once (WTO feature).

$$\phi = \alpha \, (\epsilon_i)_{i \in I_m} \, A \, (\psi_i)_{i \in I_u}$$

The Revision register follows two guidelines:

- Registers are mandatory 32-bits wide
- Read-Only reserved bit-fields have a mandatory description *Reads return 0's*.

The piece of code below takes into consideration this couple of guidelines.

```
all RegisterWidth is
  width 32 bit;
end RegisterWidth;
all ReservedROdescription is
  description ``Reads return 0's.'';
end ReservedROdescription;
ro REVISION extends RegisterWidth is
  description "This register contains
      the TP revision code";
   offset 0x0;
  ro reserved extends ReservedROdescription is
      offset 8 bit;
      reset 0x0;
      width 24 bit;
  end reserved;
  ro Rev is
      description "IP revision";
      offset 0x0;
      ro Minor is
         description "Minor Revision";
         offset 0 bit;
         reset 0x1;
         width 4 bit;
      end Minor;
      ro Major is
         description "Major Revision";
         offset 4 bit;
         reset 0x0;
         width 4 bit;
      end Major;
   end Rev;
end REVISION;
```

Hence, guidelines or pieces of common code can be written once to avoid as much as possible discrepancies and permit in the future swift updates impacting all the shared piece of code.

E.g. "*Reads return 0's.*" would be better written "*Read returns 0.*". It can be changed within the entire design just once and for all through the attribute *description* of the form *ReservedROdescription*.

Xxxxx capture front-end do not allow this kind of abstraction.

1.3 Abstraction

In order to strengthen the <u>WTO</u> hypothesis introduced above with the inheritance feature, we noticed that some piece of code are close and differ one each other in values. As standard programming languages define functions or procedures with parameters, we define *template forms* which are basically forms as defined above, but have some generic parameters $(\tau_i)_{i \in I_n}$ used within. These parameters will be instantiated during the computation steps (see relative model of computation section).

$$\phi = (\tau_i)_{i \in I_n} \alpha (\epsilon_i)_{I_m} A (\psi_i)_{i \in I_u}$$

The revision register can thus be written in a most abstract way:

all RegisterWidth is width 32 bit; end RegisterWidth; template < MAX, MIN > all ReservedRO is description ''Reads return 0's.''; offset MIN bit; width MAX - MIN + 1 bit; reset 0x0; end Reserved; template < MAX, MIN > all Beach is offset MIN bit; width MAX - MIN + 1 bit; end Beach; ro REVISION extends RegisterWidth is description "This register contains the IP revision code"; offset 0x0; ro reserved extends ReservedRO < 31, 8 > end reserved; ro Rev is description "IP revision"; offset 0x0; ro Minor extends Beach < 3, 0 > isdescription "Minor Revision"; reset 0x1; end Minor; ro Major extends Beach < 7, 4 > is description "Major Revision"; reset 0x0; end Major; end Rev; end REVISION;

We noticed that the code to be written by the designer has been dramatically reduced ensuring, as a side-effect, a decrease of possible errors. Moreover all reusable piece of code can be embedded in libraries shared by all designers. This provides for a high degree of reuse.

1.4 Constrained Abstraction

We have introduced all the needed material to capture registers efficiently. However, we face a lack of typecheck which can be very dangerous in terms of capture, and can be easily statically checked.

In the example below, the abstract parameter *RANGE* of the template form *ReservedRO* is ought to be a form since it is used within an inheritance statement. However, the designer wants the template form to get instantiated with a *Beach* form to correctly set *offset* and *width* attributes as depicted in the *Beach* template form above. This feature aims to constrain the form to get instantiated with a *Beach*-derived form. Thus, we introduce *constrained template forms*.

$$\phi = (\chi_i \to \tau_i)_{i \in I_n} \alpha (\epsilon_i)_{I_m} A (\psi_i)_{i \in I_u}$$

 χ_i is a form in Γ which ensure that the given generic

2004 © Symposium on IC Design Verifi cation

September 2004 - Dallas, TX

parameter τ_i must be χ_i -typed when expanded. Our register tiny example is hence written:

```
all RegisterWidth is
  width 32 bit;
end RegisterWidth;
template < VALUE >
all Reset is
  reset VALUE;
end Reset;
template < MAX, MIN >
all Beach is
  offset MIN bit;
  width MAX - MIN + 1 bit;
end Beach;
template < Beach -> RANGE >
all ReservedRO extends RANGE, Reset<0x0> is
  description ``Reads return 0's.'';
end Reserved;
ro REVISION extends RegisterWidth is
  description "This register contains
      the IP revision code";
   offset 0x0;
  ro reserved extends ReservedRO
      < Beach<31, 8> >
   end reserved;
   ro Rev is
      description "IP revision";
      offset 0x0;
      ro Minor extends Beach
         < 3, 0 >, Reset<0x1> is
         description "Minor Revision";
      end Minor;
      ro Major extends Beach
         < 7, 4 >, Reset<0x0> is
         description "Major Revision";
      end Major;
   end Rev;
end REVISION
```

2 Model of Computation

The proposed Model of Computation aims to translate a spread register captured system into a flat full instantiated one. Basically,

$$\phi \in \Gamma_{\phi} \to^* ! \phi \in \Phi$$

This MoC is composed of the instantiation major steps defined below. Some expansion and reduction rules are used and accurately defined in the draft dedicated to describing the mathematical model.

If the reader is interested in a full description of the semantics of the model, he can refer to the specific draft [*XXBBRDSP04*].

The instantiation is composed of two features:

1. the *inheritance* step which aims to flat all extension forms $(\epsilon_i)_{i \in I_m}$ computing access type and attributes and handle the attribute overload

September 2004 - Dallas, TX

2. the *generic instantiation* step which aims to instantiate generic parameters of template forms with values set as parameters in the caller

The *Revision* register is instantiated into its fully reduced normal form !*Revision*:

```
ro REVISION is
  description "This register contains
     the IP revision code";
  offset 0x0;
  width 32 bit;
  ro reserved is
      description "Read return 0's";
      offset 8 bit;
     reset 0x0;
     width 24 bit;
  end reserved;
  ro Rev is
     description "IP revision";
      offset 0x0;
      ro Minor is
         description "Minor Revision";
         offset 0 bit;
         reset 0x1;
         width 4 bit;
      end Minor;
      ro Major is
         description "Major Revision";
         offset 4 bit;
         reset 0x0;
         width 4 bit;
      end Major;
   end Rev;
end REVISION;
```

The given instantiated register is hence fully instantiated and checked. Therefore, if an upper form instantiates it, it will not need to be checked once again in order to be instantiated.

3 Properties

Some properties can be statically checked to ensure that the register database is coherent.

3.1 Contiguity coherency

If a form declares a fixed width through the *width* attribute, hence all bits are expected to be defined. E.g. the *Revision* register declared a 32-bit wide range and defined three exclusive ranges: a 24-bit range for the reserved bit-field, a 4-bit range for the Major bit-field and a 4-bit range for the Minor bit-field.

3.2 Range coherency

Ranges are defined with the offset attribute. Their width is infered, if possible, through the width defined within the hierarchal inner forms. They must remain exclusive one each other.

The property of range coherency assesses that the refined inner forms meet the contiguity coherency property.

However, this property does not assess that all bits are defined.

3.3 Overlap coherency

Two defined bit-fields must remain exclusive according to the contiguity coherency property. However, sometimes, a same bit-field could have different behavior according to an activation condition. Two kinds of form can be exhibited:

- 1. *dual forms* which are final forms only tuning the access type and the reset value, without any activation condition.
- 2. *modal forms* which closely depend on activation condition $\beta \in \mathfrak{M}$ and allow the user to write two separate behaviors in the same form range.

4 Example: IRQ management module

An IRQ management module is composed of IRQ lines which will be enabled or disabled by hardware components. These lines can be read to check if the concerned IRQ has occurred and reset according to 4 commonly used protocols — read / write 1 to clear, read / write 0 to clear, read / write 1 to set, read / write 0 to set.

For our purpose, the truth table below depicts the behavior of a read / write 1 to clear IRQ line.

In this example, we assume that no bypass protocol is defined i.e. an IRQ arisen by a component and a read/write command cannot occur at the meantime. The figure below highlights a basic IRQ line management.



wire	IRQ	command	IRQ
0	0	idle	0
0	1	idle	0
1	0	idle	1
1	1	idle	1
0	0	read	0 (Read0x0)
0	0	write 1 (Write0x1)	0
0	0	write 0 (Write0x0)	0
0	1	read	1 (Read0x1)
0	1	write 1 (Write0x1)	0
0	1	write 0 (Write0x0)	1

4.1 Shared library: Generic read / write 1 to clear form declaration

This form is basically fully instantiated yet:

```
rw IRQ_rwltoClr is
  description ``IRQ line R/W 1 to clear'';
  reset 0x0;
  width 1 bit;
  ro FalseEvent is
      description "The event is false";
      reset 0x0;
   end FalseEvent;
   ro PendingEvent is
      description "The event is true (pending)";
      reset 0x1;
   end PendingEvent;
   wo UnchangedStatus is
      description "The event status bit unchanged";
      reset 0x0;
   end UnchangedStatus;
   wo ResetStatus is
      description "The event status bit is reset";
      reset 0x1;
   end ResetStatus;
end IRQ_rwltoClr;
```

However, the reset value of this 1-bit register is hard coded with 0. In some cases, this value should be set to 1. The first solution would be to manually duplicate this capture to hard code the reset value to 1. The second solution, highly recommended, is to give a parameter to this form. This fully instantiated form in Γ_{ϕ} is abstracted and goes in $\Gamma_{<\phi>}$ in order to be instantiated with the wanted reset value.

```
template < RESET ;
rw IRO rw1toClr is
  description ``IRQ line R/W 1 to clear'';
  reset RESET;
  width 1 bit;
   ro FalseEvent is
      description "The event is false";
      reset 0x0;
  end FalseEvent;
  ro PendingEvent is
     description "The event is true (pending)";
      reset 0x1;
  end PendingEvent;
   wo UnchangedStatus is
      description "The event status bit unchanged";
      reset 0x0;
   end UnchangedStatus;
```

2004 © Symposium on IC Design Verifi cation

September 2004 - Dallas, TX

```
wo ResetStatus is
    description "The event status bit is reset";
    reset 0x1;
    end ResetStatus;
end IRQ_rwltoClr;
```

4.2 Instantiation: Declaration of an IRQ management module

We are designing a FIFO which has 12 IRQ lines. We will declare and define the module which aims to manage these lines.

The current specification defines 5 IRQs as depicted in the figure below.



We can notice that 7 bits, in the range [5, 11], are reserved for future usage. These bits are hard-wired connected to return 0's on read and discard any writing values. We will therefore define a template form which declares such a range of reserved bits.

```
template < N, OFFSET >
rw Reserved is
  description ``Reserved. Reads return 0's'';
  reset 0x0;
  width N bit;
  offset OFFSET bit;
end Reserved;
root all FIFO_IRQs is
  description '`FIFO IRQ lines management'';
  width 12 bit;
  all reserved extends Reserved< 12 - 5, 5 >
  end reserved;
  all FIFO_UF extends IRQ_rwltoClr<0> is
      description ``FIFO UnderFlow IRQ'';
      offset 0x0;
  end FIFO UF;
  all FIFO_EPTY extends IRQ_rwltoClr<1> is
      description ``FIFO Empty IRQ'';
      offset 1 bit;
  end FIFO_EPTY;
  all FIFO_THR extends IRQ_rwltoClr<0> is
      description ``The threshold in the FIFO
        has been reached'';
      offset 2 bit;
  end FIFO_THR;
  all FIFO_FULL extends IRQ_rwltoClr<0> is
      description ``The FIFO is Full'';
      offset 3 bit;
  end FIFO_FULL;
  all FIFO_OF extends IRQ_rwltoClr<0> is
      description ``FIFO OverFlow IRO'';
      offset 4 bit;
  end FIFO OF;
end FIFO_IRQs;
```

4.3 MoC: Full instantiated IRQ management component for the FIFO

Hence,		C	our		system
\mathfrak{P}_{FIFO_IRQs}		=	$(\Gamma,$	$FIFO_{-}$	IRQs),
Γ_{ϕ}	=	{	FI	FO_IRQ	$s $ },
$\Gamma_{<\phi>} =$	$\{ Re$	eserved,	IRQ_{-}	rw1 to Clr	} can
be computed	to get	reduced	into !	FIFO_II	RQs as
depicted by th	e follo	wing tree.	Our co	ompacted	50 lines
description, is	comp	uted into	a RDO	outputted	l flatten
file, 115 lines	wide,	fully RD c	complia	nt. Conse	quently,
as soon as FIL	FO_I	$RQs \in \Gamma_{q}$		iced into a	normal
form $!FIFO_$	IRQs	, we can u	ise it as	a core con	nponent
avoiding costly	y reduc	ctions.			



```
offset 0x0;
width 12 bit;
rw reserved is
  description "Reserved. Reads return 0's";
   offset 5 bit;
  reset 0x0;
   width 7 bit;
end reserved;
rw FIFO OF is
  description "FIFO OverFlow IRQ";
   offset 4 bit;
  reset 0;
   width 1 bit;
   ro FalseEvent is
      description "The event is false";
      reset 0x0;
   end FalseEvent;
   ro PendingEvent is
      description "The event is true (pending)";
      reset 0x1;
   end PendingEvent;
   wo UnchangedStatus is
      description "The event status bit unchanged";
      reset 0x0;
   end UnchangedStatus;
   wo ResetStatus is
      description "The event status bit is reset";
      reset 0x1;
   end ResetStatus;
end FIFO_OF;
```

```
rw FIFO FULL is
   description "The FIFO is Full";
   offset 3 bit;
   reset 0;
   width 1 bit;
   ro FalseEvent is
      description "The event is false";
      reset 0x0;
   end FalseEvent;
   ro PendingEvent is
  description "The event is true (pending)";
      reset 0x1;
   end PendingEvent;
   wo UnchangedStatus is
      description "The event status bit unchanged";
      reset 0x0;
   end UnchangedStatus;
   wo ResetStatus is
description "The event status bit is reset";
      reset 0x1;
   end ResetStatus;
end FIFO_FULL;
rw FIFO_THR is
   description "The threshold in the FIFO
      has been reached";
   offset 2 bit;
   reset 0;
   width 1 bit;
   ro FalseEvent is
      description "The event is false";
      reset 0x0;
   end FalseEvent;
   ro PendingEvent is
      description "The event is true (pending)";
      reset 0x1;
   end PendingEvent;
   wo UnchangedStatus is
      description "The event status bit unchanged";
      reset 0x0;
   end UnchangedStatus;
   wo ResetStatus is
      description "The event status bit is reset";
      reset 0x1;
   end ResetStatus;
end FIFO_THR;
rw FIFO_EPTY is
   description "FIFO Empty IRQ";
   offset 1 bit;
   reset 1;
   width 1 bit;
   ro FalseEvent is
      description "The event is false";
      reset 0x0;
   end FalseEvent;
   ro PendingEvent is
      description "The event is true (pending)";
      reset 0x1;
   end PendingEvent;
   wo UnchangedStatus is
      description "The event status bit unchanged";
      reset 0x0;
   end UnchangedStatus;
   wo ResetStatus is
    description "The event status bit is reset";
      reset 0x1;
   end ResetStatus;
end FIFO_EPTY;
rw FIFO UF is
   description "FIFO UnderFlow IRQ";
   offset 0x0;
   reset 0;
   width 1 bit;
   ro FalseEvent is
      description "The event is false";
      reset 0x0;
   end FalseEvent;
   ro PendingEvent is
```

```
description "The event is true (pending)";
    reset 0x1;
end PendingEvent;
wo UnchangedStatus is
    description "The event status bit unchanged";
    reset 0x0;
end UnchangedStatus;
wo ResetStatus is
    description "The event status bit is reset";
    reset 0x1;
end ResetStatus;
end FIFO_UF;
end FIFO_IRQs;
```

4.4 Reuse: Component managing the IRQs of a module

This section aims at describing how to easily use the IRQ management component defined above as a part of a top component registers description using a FIFO, but does not focus on a full description.

Be the following piece of registers specification of the Camera Core module. All registers are 32-bit wide.

Register	Offset	Description
CC_REVISION	0x00	Revision Register
CC_IRQSTATUS	0x18	Interrupt Status Register
CC_IRQENABLE	0x1C	Interrupt Enable Register
CC_FIFODATA	0x4C	FIFO Data Register
CC_TEST	0x50	Test Register

The piece of specification below of the CC_IRQSTATUS register depicts the ranges affected for the three main sub-modules of the Camera Core IP.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	nadiacai		Sei IR(ria Qs	ĺ		1	ese	rvei	d		F	'ar IR	all Qs	el	f	ese	rvei	đ				F	IF	0	IF	S	s			

/* the file includes the template form Reserved */ include ''common.rd''

/* the fully instantiated form FIFO_IRQs is imported */ import ''FIFO_IRQs''

/* The form should have got declared in the file common.rd */
all XXXXxxxRegisterWidth is
width 32 bit;

end XXXXxxxxRegisterWidth;

template < RESET, N, OFFSET >
ro ReservedDoNotWrite extends Reserved<N, OFFSET > is
 description ``Reserved. Do Not Write'';
 reset RESET;
end ReservedDoNotWrite;

all IRQSTATUS extends XXXXxxxxRegisterWidth is description ``Interrupt Status Register'';

September 2004 - Dallas, TX

```
all FIFO extends FIFO IROs is
      offset 0x0;
   end FIFO;
  all reserved extends Reserved
      <16 - 12, 12>
   end reserved;
  all PARALLEL ... end PARALLEL;
  all reserved extends Reserved
      <26 - 20, 20>
   end reserved;
  all SERIAL ... end SERIAL;
  all reserved extends ReservedDoNotWrite
      <0b11, 2, 30>
  end reserved;
end IROSTATUS;
main all CameraCore is
  description ''Camera Core Registers'';
   ro CC_REVISION ... end CC_REVISION;
  rw CC_IRQSTATUS extends IRQSTATUS is
     offset 0x18;
   end CC_IRQSTATUS;
  rw CC_IRQENABLE ... end CC_IRQENABLE;
  rw CC_FIFODATA ... end CC_FIFODATA;
                   ... end CC_TEST;
   ro CC_TEST
end CameraCore;
```

5 Purposed targeted outputs

As aforesaid, we can generate some code from the unique checked registers database, according to the targets. This section shows some piece of outputted code coming from the underflow IRQ.

5.1 C check library

Xxxx generates a C library especially focusing on register check. The piece of code below highlights the basic tests given to the validation phase. Algorithms implemented within each function are given by XX according to the needs. We especially want to check:

- **power-on-reset values**, checked when the system is powered up to ensure that the read value coming from the RTL is equal to the one foreseen by the spec owner.
- **access types**, checked to ensure that a read-only register cannot be written, or that a read-write register can genuinely get written
- **custom access type** like read-write-1toClear to ensure that the RTL is aligned with this special behavior. The example below depicts this case.

```
void CameraCoreRegisterIntegrityTest
  (UWORD32 baseAddress);
void CameraCoreRegisterIntegrityRW0ToSetTest
  (UWORD32 baseAddress);
void CameraCoreRegisterIntegrityRW1ToSetTest
  (UWORD32 baseAddress);
void CameraCoreRegisterIntegrityRW0ToClrTest
```

September 2004 - Dallas, TX

(UWORD32 baseAddress); void CameraCoreRegisterIntegrityRW0ToClrTest (UWORD32 baseAddress); void CameraCoreRegisterIntegrityRW1TogPerBitTest (UWORD32 baseAddress);

5.2 C library

Xxxx generates C library to allow validation user to read and write registers directly using an API. General header files contain the base addresses of each IP. The end-user can therefore easily include in his hand-written test-case the header file containing the base addresses of the wanted view: for XXXXxxxx, these basic views are either ARM11 or DSP.

```
[...]
#define CC IROSTATUSFIFOFIFO UFRead32(baseAddress)\
 (_DEBUG_LEVEL_1_EASI(\
   EASIL1_CC_IRQSTATUSFIFOFIFO_UFRead32), \
  (((RD_MEM_32_VOLATILE(((UWORD32)(baseAddress)))
      +(CC_IRQSTATUS_FIFO_OFFSET))) &\
     CC_IRQSTATUS_FIFO_FIFO_UF_MASK) >>\
     CC_IRQSTATUS_FIFO_FIFO_UF_OFFSET))
[...]
#define CC_IRQSTATUSFIFOFIFO_UFWrite32(\
 baseAddress,
 value
\{ \setminus
const UWORD32 offset = CC_IRQSTATUS_FIFO_OFFSET;\
register UWORD32 data = RD_MEM_32_VOLATILE(
  ((UWORD32)(baseAddress))+offset);
register UWORD32 newValue = ((UWORD32)(value));\
_DEBUG_LEVEL_1_EASI(\
   EASIL1_CC_IRQSTATUSFIFOFIFO_UFWrite32);\
data &= ~(CC_IRQSTATUS_FIFO_FIFO_UF_MASK);\
newValue <<= CC_IRQSTATUS_FIF0_FIF0_UF_OFFSET; \
newValue &= CC_IRQSTATUS_FIFO_FIFO_UF_MASK; \
newValue |= data;\
WR_MEM_32_VOLATILE((UWORD32)(baseAddress)\
  + offset, newValue);\
[...]
```

The piece of code above defines a C-ANSI macro to read and write a 32-bit wide FIFO data. The base address must be given as parameter since separate header files can be included, each containing a different base address according to the desired view. We can notice that the level of hierarchy is kept through the macro identifier, even if it lacks of readability.

This effective lack of readability led on a pure hierarchal approach prototype, taking advantages from the syntax of the C language.

5.3 Hierarchal C tree

Indeed, C has a *struct* statement which is used to add more readability and structure. This structured view

 $2004 \ensuremath{\textcircled{O}}$ Symposium on IC Design Verifi cation

is built over the approach presented above to link the <u>read/write</u> field to the right flat function. The piece of code below depicts the header file to be included before using the tree. For example, the simplest way to clear the FIFO_UF interrupt is:

 $CameraSS.CameraCore.CC_IRQSTATUS.FIFO_IFRO_UF.ResetStatus.write()$

This hierarchal tree is very accurate and genuinely fit the generated relative documentation, since the source file is unique for this couple of generated purposed targets.

```
struct CameraSS_struct {
  struct {
    struct {
      struct {
        UWORD32 (*read)(void);
        void (*write)(UWORD32);
        [...]
        struct
          UWORD32 (*read)(void);
          void (*write)(UWORD32);
          struct {
            UWORD32 (*read)(void);
          } FalseEvent;
          struct
            void (*write)(void);
          } UnchangedStatus;
          struct {
            UWORD32 (*read)(void);
          } PendingEvent;
          struct {
            void (*write)(void);
          } ResetStatus;
        } FIFO_UF;
      } FIFO;
    } CC_IRQSTATUS;
    CameraCore;
} CameraSS;
```

5.4 E code

Xxxx currently generates E code to check the registers with XxxXxx, according to custom XX features. Here is thus highlighted a major feature of this approach: as soon as the data are captured once and for all in a unique database, we have the possibility to run a custom generator. In the future we will be able to write ourselves our own tactical generators.

5.5 Documentation output

This section is especially devoted to TRM teams. The outputted file is ought to be read by humans, formatted according to XX guidelines driven by publishing rules. This kind of outputted format can be RTF/DOC (MS-Word application), PDF/PostScript, HTML or other custom formats like InterLeaf or FrameMaker.

For our prototyping purposes, this feature has been addressed through LATEX2e, using an IEEE layout package, to generate PostScript, PDF or any other humanoriented format taking LATEXOr PDF as input.

Xxxxx xx is able to generate an high quality RTF which was used within the final TRM.

5.6 Hierarchal Flat RD-XML

The aim of this subsection does not focus on the presentation of forte and weaknesses of the XML format. We propose below a RD grammar over XML. The reader interested in getting more information about the basic XML format is highly invited to check out the W3C organization web-site [*W3CWS*].

This format is especially dedicated to exchange data and furthermore well-suited to

- handle gateways to import register captured data coming from an another register formalism (e.g. Xxxxx xx XML into RD)
- handle gateways to import legacy code written to capture registers in the past (e.g. ad-hoc XLS spreadsheet into RD)
- 3. handle gateways to export register-oriented data (e.g RD to Xxxxx RTL.conf, RD to XxxXX)
- 4. handle data capture through XML editors from the market (e.g. XMLSpy, Morphon)

The piece of code below is generated for our trial purposed crude example. We can notice that the tags are really close to the pure RD keywords. We mustn't be astonished since XML can be seen as a front-end standardized format, becoming a language over XML as soon as a semantics is given to each tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Forms PUBLIC "" "[...]/rd.dtd">
<Forms>
  <InstanceForm name="CameraSS" access="RW">
    <InnerForm name="CameraCore" access="RW">
      <offset>
        <Hex value="0"/>
      </offset>
      <InnerForm name="CC IROSTATUS" access="RW">
        <offset>
          <Hex value="0"/>
        </offset>
        <InnerForm name="FIFO" access="RW">
          <description>
            <![CDATA[FIFO IRQ lines management]]>
          </description>
          <offset>
            <Hex value="18"/>
          </offset>
```

September 2004 – Dallas, TX

```
<width>
            <BitUnit>
              <Dec value="32"/>
             </BitUnit>
          </width>
          <InnerForm access="RO" reserved="true">
            <offset>
              <BitUnit>
                <Dec value="12"/>
              </BitUnit>
            </offset>
            <width>
              <BitUnit>
                <BinaryExpressionPlus>
                  <BinaryExpressionMinus>
                     <Dec value="31"/>
                     <Dec value="12"/>
                  </BinaryExpressionMinus>
                  <Dec value="1"/>
                </BinaryExpressionPlus>
              </BitUnit>
            </width>
            <reset>
              <Hex value="0"/>
            </reset>
            </InnerForm>
            [...]
            <InnerForm name="FIFO_UF" access="RW">
              <description>
                <![CDATA[FIFO UnderFlow IRQ]]>
              </description>
              <offset>
                <Hex value="0"/>
              </offset>
              <width>
                <BitUnit>
                  <Dec value="1"/>
                </BitUnit>
              </width>
              <reset>
                <Dec value="0"/>
              </reset>
              <InnerForm name="FalseEvent" access="RO">
                <description>
                  <![CDATA[The event is false]]>
                </description>
                <reset>
                  <Hex value="0"/>
                </reset>
              </InnerForm>
              [...]
            </InnerForm>
          </TnnerForm>
        </InnerForm>
      </TnnerForm>
   </InnerForm>
  </InstanceForm>
</Forms>
```

```
# ## FIFO IRQ lines management
register FIFO {
    param access_type rw
    param reset_value 0x2
    param base_address 0x18
    param rw_mask 0x1F
    param data_width 0x20
}
```

5.8 XXXXXXXXX XXXXXXX for debug

Xxxxxxxx is a famous german tool aiming to debug hardware. Registers can be read and written according to their access type thru a software GUI to program a hardware platform connected to the computer. The piece of file below shows how to program the GUI to declare the FIFO register and its inner bit-fields.

```
tree "CameraSS"
 tree "CameraCore"
 base 0x0
  tree "CC IROSTATUS"
   base 0x0
     tree "FIFO: FIFO IRQ lines management"
      group 0x18--0x37
      line.long 0x0 "value ,FIFO IRQ lines management"
        textline ""
       bitfld.long 0x0 4. "FIFO_OF ,FIFO OverFlow IRQ"
          "FalseEvent, PendingEvent"
        textline "
       bitfld.long 0x0 3. "FIFO_FULL ,The FIFO is Full"
          "FalseEvent, PendingEvent"
        textline "
        bitfld.long 0x0 2. \tt "FIFO\_THR ,The threshold in
          the FIFO has been reached"
          "FalseEvent, PendingEvent
        textline ""
        bitfld.long 0x0 1. "FIFO_EPTY ,FIFO Empty IRQ"
          "FalseEvent, PendingEvent"
        textline "
        bitfld.long 0x0 0. "FIFO_UF ,FIFO UnderFlow IRQ"
          "FalseEvent, PendingEvent"
  tree.end
 tree.end
tree.end
tree.end
```

6 Results

5.7 Gateway to Xxxxxx rtl.conf

Xxxxx rtl.conf configuration files aim to capture OCP data to automatically generate HDL and test patterns. Some registers especially defined in the standard and XX guidelines for the OCP buses programming are already captured in any RD-oriented language. Xxxxxx added extensions to their proprietary format to also capture the other registers.

```
# root node 'CameraSS' for registers dump
```

September 2004 - Dallas, TX

On a real hands-on example dedicated to highlight the main features of this high level of registers' capture, we got the following figures:

modules	number	lines (RD)	lines (Xxxxx xx)
shared	23	303	not handled
standalone	8	163	GUI captured
computed	1	983	1676

Remarks

- basically RD capture and Xxxxx xx capture cannot be compared since in RD the capture is performed with a textual editor, whereas with Xxxxx the capture is handled with a graphical editor. We can notice that with the textual editor, the useful time is the one needed to enter the relevant information which is the same as that captured with the graphical editor, with an overhead in terms of keywords. With the Xxxxx xx graphical editor the overhead is composed of the time needed to move the mouse and click on the different glyphs, and the time needed to capture the shared information more than once.
- this example depicts a fictitious Camera Sub-System using an OCP shared library and two modes, one for the Debug, and the other one for the Regular behavior. It was developed to highlight and to strengthen features.
- the Xxxxx xx was generated with a translator taking as input a RD description, following translation rules between the high level Register Description format and Xxxxx xx XML.

Summary

We have exposed how important it is to manage registers through a unique database to avoid discrepancies between the TRM, the validation and other potential targets such as the RTL. Write-things-once leads on unify from the specification, all possible targets, saving valuable resources.

We have also introduced a high level Register Description format called RD to capture the main features needed to work with registers. We have thus prototyped translators, bridges between tools, different capturing ways, and discussed a model of computation which aims to strengthen the reuse feature, especially with shared libraries.

Xxxx toolset offers a way to handle registers in this direction. This approach was widely used to maintain the XXXXxxx registers and has encountered a real success. However, this kind of tool involves several people working on different stages of the flow, in different world-wide sites, and is therefore ought to be driven with strong processes and associated methodologies.

References

- [XXBxx504] Bertrand Blanc, Xxxxx xx, a subset of Register Descritpion Definition, XX internal document, 2004
- [*W3CWS*] World Wide Web Consortium web-site, *http://www.w3.org*
- [BBRDSP04] Bertrand Blanc, Register Description Semantics Proposal, XX internal document, 2004

[xxxxWS] Xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx